Model-Based Testing with Graph Transformation Systems

Vincent de Bruijn

January 6, 2012

Abstract

This report describes the setup of a research project where the use of *Graph Transformation Systems* (GTS) in model-based testing is researched. The goal of the project is to create a system for automatic test generation on GTSs and to validate this system. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as part of the system. The system will be validated using the results of several case-studies. GTSs have many structural advantages, which are potential benefits for the model-based testing process.

Contents

1 Introduction

Limited time and resources in software projects often make testing a neglected part of the software development process. However, testing is an important part of software development, because it decreases future maintainance costs. In other cases, a lot of time and resources are spent on the testing process, which makes the overall development process more expensive. Therefore, the testing process should be as efficient as possible.

Hier komt nog een citation naar een onderzoek over besteedde tijd in testing en de kosten

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed A widely used practice is maintaining a *test suite*, which is a collection of tests. However, when the creation of a test suite is done manually, this still leaves room for human error. Also, manual creation of test-cases is not time-efficient.

Creating an abstract representation or a *model* of the system is a way to tackle these problems. What is meant by a model in this report, is the description of the behavior of a system. Models such as state charts and sequence charts, which only describe the system architecture, are not considered here. A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. This leads to a larger test suite in a shorter amount of time than if done manually. These models are created from the specifications from the software developer, end-user and/or owner. The process of collecting the specifications by the tester into a model is still error-prone, due to imprecise, incomplete and ambiguous communication of those specifications. If the tester makes a wrong interpretation, the constructed model becomes incorrect.

Hier moet nog een citation komen over requirements analysis en de problemen daarbij

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which obstructs this feedback process. Models that are often used are state machines, i.e. a collection of nodes representing the states of the system connected by transitions representing an action taken by the system. The overview is quickly lost with models of this type. The problem is therefore that errors in such models are not easily detected.

A formalism that among other things can describe software systems is the Graph Transformation System (GTS). These systems express the system state and transitions by means of graphs. These graphs are a representation of the start state and the behavior of the system. The model is therefore a collection of graphs, each of a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling. Graph transformations and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]: "Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples." An informative paper on graph transformations is written by Heckel et al.[7]. A quote from this paper: "Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular."

1.1 Research setup

This report describes the setup of a project where the use of GTSs in model-based testing is researched. In the introduction the motivation is given for using this type of modelling technique. The goal is to create a system for automatic test generation on GTSs. If the assumptions that GTSs provide a more intuitive modelling and testing process hold, this new testing approach will lead to a more efficient testing process and fewer incorrect models. The system design, once implemented and validated, provides a valuable contribution to the testing paradigm.

Tools that perform statespace exploration on GTSs and tools for automatic test generation already exist. Two of these tools will be used to perform these functions. The graph transformation tool GROOVE[1] will be used to model and explore the GTS. The testing tool developed by Axini[2] is used for the automatic test generation. This tool has no name, but will be referred to in this report as Axini Test Manager (ATM).

1.2 Research questions

The research questions are split into a design and validation component:

1. **Design**: How can automatic test generation be done using graph transformation systems? In particular, how can ATM be used together with GROOVE to achieve this?
2. **Validation**: What are the strengths and weaknesses of using graph transformation systems in model-based testing?

The result of the design question will be one system which incorporates ATM and GROOVE. This system will be referred to as the GROOVE-Axini Testing System (GRATiS). In order to answer the first research question sufficiently, GRATiS should produce tests on the basis of a GTS and give correct verdicts whether the system contains errors or not.

The criteria used to assess the strengths and weaknesses are split into two parts: the objective and the subjective criteria. The objective criteria are the measurements that can be done on GRATiS, such as speed and statespace size. The subjective criteria are related to how easy graph transformation models are to use and maintain. The assessment of the latter criteria requires a human component. The criteria and methods for the assessment are elaborated in section 6.2.

GRATiS will be validated using case-studies. These case-studies are done with existing specifications from systems tested by Axini. Each case-study will have a GTS and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the model and the test process will then be compared as part of the validation. This includes the criteria mentioned in section 1.2. This is explained in more detail in section 6.2.

1.3 Structure of the report

The structure of this document is as follows: the general model-based testing process is reviewed in section 2. Sections 3 and 4 describe two relevant models for this research project. The tools used are introduced in section 5. The methods used in the research are in section 6. Finally, a summary is given in section 7. The planning for the research project is in appendix A.

2 Model-based Testing

hier moeten nog even een consistent verhaal van gemaakt worden.

Formal testing theory was introduced by De Nicola et al.[13]. This testing theory was first used in algorithms for automatic test generation by Brinksma [3]. The specification language LOTOS was used in this research as a defining notation for transition systems. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [16]. A good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [11]. A tool for the automatic synthesis of test cases for nondeterministic systems is TGV [8].

[1] http://sourceforge.net/projects/groove/
[2] http://www.axini.nl/

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted in Figure 1. The specification, e.g. a model of the system, is given to a test derivation component which generates test cases. These test cases are passed to a component that executes the test cases on the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates the test cases, the stimuli and the responses and gives a 'pass' or a 'fail' verdict whether the SUT conforms to the specification or not. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state. Test cases should always allow reaching a pass or fail state within finite time.

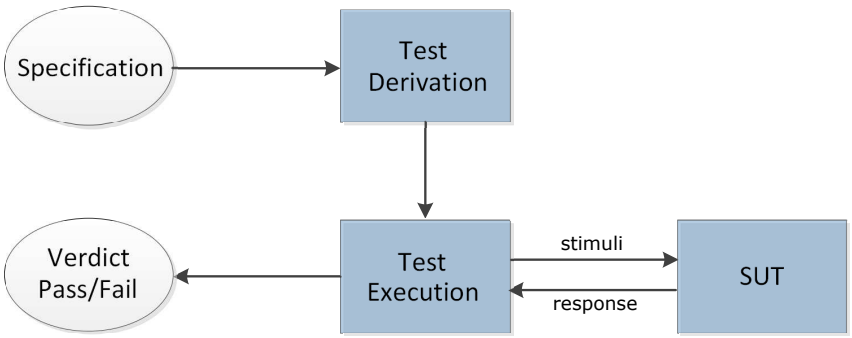


Figure 1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on the fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 2. A tool for on-the-fly testing is TorX [15], which integrates automatic test generation, test execution, and test analysis. A version of this tool written in Java under continuous development is JTorX [2].

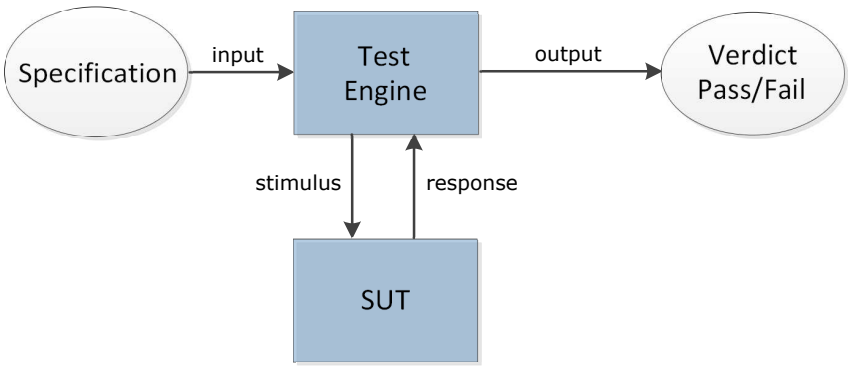


Figure 2: A general 'on-the-fly' model-based testing setup

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data. First, two types of models are introduced. These are basic formalisms useful to understand the models in the rest of the paper. Then, the notion of *coverage* is explained.

2.1 Labelled Transition Systems

A labelled transition system is a structure consisting of states with transitions, labelled with actions, between them. The states model the system states; the labelled transitions model the actions that a system can perform.

2.1.1 Definition

A labelled transition system is a 4-tuple $Q, L, T, q0$, where:

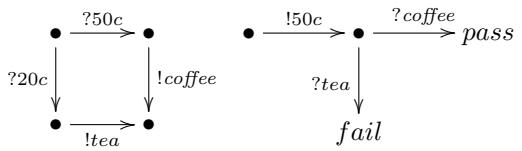
- Q is a finite, non-empty set of states
- L is a finite set of labels
- $T \in Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation
- $q0 \in Q$ is the initial state.

We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, , q') \in T$. The informal idea of such a transition is that when the system is in state q it may perform action μ, and go to state q'.

2.2 Input-Output Transition Systems

A useful transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans[17]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. These are similar to LTSs with the exception that labels have a type. This type can be 'input' or 'output'.

An example of such an IOTS is shown in Figure 3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a question mark, the outputs are preceded by an exclamation mark. This system is a specification of a coffee machine. A test case can also be described by an IOTS. A test case for the coffee machine is given in Figure 3b. The test case is derived from the coffee machine IOTS in the Test Derivation component. The Test Execution component applies the stimulus '50c' to the SUT. When the SUT responds with 'tea', the test case fails and when the SUT responds with 'coffee', the test case passes.



(a) An IOTS (b) An IOTS test case

Figure 3: The specification of a coffee machine and a test case as an IOTS

Unnecessary nondeterminism should be avoided in test cases [18]: "In the first place, this implies that the test case itself is deterministic. In the second place, this means that a tester should never offer more than one input action (from the perspective of the implementation) at a time. Since the implementation is able to accept any input action, offering more inputs would always lead to an unnecessarily non-deterministic continuation of the test run. Having a deterministic test case does not imply that a test run has a unique result: due to non-determinism in the implementation under test, and due to non-determinism in the test run itself, the repetition of a test run may lead to a different result".

2.3 Coverage

The amount of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time spent testing. Coverage statistics help with test selection. When the SUT is a black-box, typical coverage metrics are state and transition coverage of the specification [10, 12, 6].

Now the coverage metrics of the IOTS test case example in 3b can be calculated. The test case tests one path though the specification. The state coverage is therefore 75% and the transition coverage is 50%.

3 Symbolic Transition Systems

Symbolic test generation is introduced by Rusu et al. [14], using Input-Output Symbolic Transition Systems (IOSTSs). Symbolic Transition Systems (STSs) are introduced by Frantzen et al. [9]. A tool that generates tests based on symbolic specification is the STG tool, described in Clarke et al. [4].

In this section, STSs are introduced which combine LTSs with a data type oriented approach. Among other tools, they are used in practice in ATM.

STSs are made of *locations* and *switch relations*. A location represents a point in the control flow of the system, but has no information on data values. A switch relation switches control from one location to another. The *location variables* are a representation of the data values in the model. They are independent of the locations of the STS. *Gates* are often called the action or label of a switch relation. Gates also have *interaction variables*, which are data values specific to the gate. Switch relations also have *guards* and *update mappings*. A guard is a boolean function, which restricts the use of the switch relation when the function evaluates to false. An update mapping is a mapping of location variables to an arithmetic function. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the evaluation of their arithmetic function. We define $\mathcal{F}(\mathcal{T})$ to be a collection of boolean functions over a set of terms $\mathcal{T}$. These terms are variables and basic data types, such as integers, strings and arrays. We define $\mathcal{G}(\mathcal{T})$ to be a collection of arithmetic functions over a set of terms $\mathcal{T}$.

3.1 Definition

The definition of an STS that follows is based on the definition given by Frantzen et al. [9]. A Symbolic Transition System is a tuple $\langle L, l_0, \mathcal{V}, \imath, \mathcal{I}, \Lambda, \rightarrow \rangle$, where:

- L is a finite set of locations and $l_0 \in L$ is the initial location.
- $\mathcal{V}$ is a finite set of location variables.
- $\imath$ is a mapping $\{v \mapsto x \mid v \in \mathcal{V}, x \in \mathcal{X}\}$ where $\mathcal{X}$ is a collection of basic data types, representing the initialisation of the location variables.
- $\mathcal{I}$ is a set of interaction variables, disjoint from $\mathcal{V}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau (\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted $arity(\lambda)$, is a natural number. The parameters of a gate $\lambda \in \Lambda_\tau$, denoted $param(\lambda)$, are a tuple of length $arity(\lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.
- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathcal{F}(\mathcal{V} \cup \mathcal{I} \cup \mathcal{X}) \times \{v \mapsto x \mid v \in \mathcal{V}, x \in \mathcal{G}(\mathcal{V} \cup \mathcal{I} \cup \mathcal{X})\} \times L$, is the switch relation. We write $l \xrightarrow{\lambda, \phi, \rho} l'$ instead of $(l, \lambda, \phi, \rho, l') \in \rightarrow$, where ϕ is referred to as the guard and ρ as the

update mapping. We require $var(\phi) \cup var(\rho) \subseteq \mathcal{V} \cup param(\lambda)$, where var is the collection of the variables used in the given guard or update mapping.

There exists a mapping from an STS to an LTS, where each location maps to a collection of states and each switch relation maps to a collection of transitions. As mentioned before, the data values represented by the location variables are not included in the locations. A state in an LTS does include the data values. Thus, a state in the collection mapped by a location is the combination of the location and an evaluation of each location variable. The switch relation from a location A to a location B maps to all transitions from the states mapped by location A to all states mapped by location B. The guard and update mapping of the switch relation determine the source and target state of each transition. The label of the transition is determined by the gate of the switch relation and an evaluation of the interaction variables. In section 5.3 the LTS is shown of the STS in the example in section 3.2.

An IOSTS can now easily be defined. The same difference between LTSs and IOTSs applies, namely each gate in an IOSTS has a type, 'input' or 'output'.

3.2 Example

In Figure 4 a simple board game is shown, where two players consecutively throw a die and move along four squares. The defining tuple of the IOSTS is:

$\langle \{throw, move\}, throw, \{T, P, D\}, \{T \mapsto 0, P \mapsto [0,2], D \mapsto 0\}, \{d, p, l\}, \{throw?, move!\},$
$\{throw \xrightarrow{throw?,1<=d<=6,D\mapsto d} move, move \xrightarrow{move!,T=p \wedge l=(P[p]+D)\%4,P[p]\mapsto l,T\mapsto p\%2} throw\}\rangle$

The variables T, P and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate $move!$ has $param = \langle p, l \rangle$ symbolizing which player moves to which location. The input gate $throw?$ has $param = \langle d \rangle$ symbolizing which number is thrown by the die. The switch relation with gate $throw?$ has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate $move!$ have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 4 the first row of text on the switch relation shows the gate, the second row shows the constraint and the last row shows the update mapping.

4 Graph Transformation Systems

A Graph Transformation System is composed of a start graph and a set of transition rules. The start graph describes the system in its initial state. The transition rules apply changes to the graph, creating a new graph which describes the system in its new state.

4.1 Graphs & Morphisms

A graph is a tuple $\langle L, N, E \rangle$, where:

- L is a set of labels
- N is a set of nodes, where each $n \in N$ has a label $l \in L$
- E is a set of edges, where each $e \in E$ has a label $l \in L$ and nodes $source, target \in N$

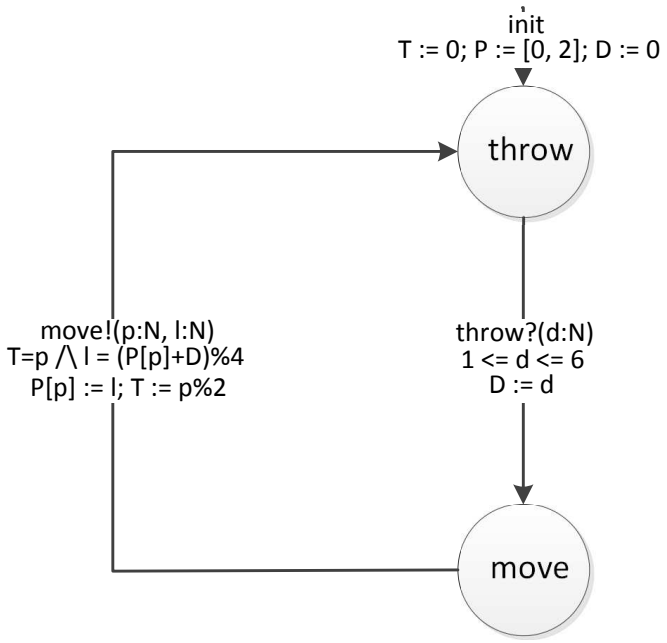


Figure 4: The STS of a board game example.

A graph H has an *occurrence* in a graph G, denoted by $L \to G$, if there is a mapping *occ* which maps the nodes and the edges of H to the nodes and the edges of G, respectively, and preserves sources, targets, and labellings.

4.2 Graph Transformation rules

A transformation rule is a tuple $\langle LHS, NAC, RHS \rangle$, where:

- LHS is a graph representing the left-hand side of the rule
- NAC is a set of graphs representing the negative application conditions
- RHS is a graph representing the right-hand side of the rule

Nodes and edges in the LHS graph have a mapping Map to nodes and edges in the RHS graph. This mapping is not defined formally, as this is out of the scope of this report. One property of the mapping is that if an element in LHS maps to an element in RHS, it must hold that their labels are the same.

ik kan de mapping ook onderdeel maken van de rule, dan 'bestaat' deze gewoon

A rule R is applicable on a graph G if its LHS has an occurrence in G and none of the graphs in its NAC have an occurrence in G. After the rule application, all elements in LHS not part of Map are removed from G and all elements in RHS not part of Map are added to G. All elements in Map are kept.

4.3 GTS Definition

A Graph Transformation System is a tuple $\langle G, R \rangle$, where:

- G is a graph of the start state
- R is a set of transformation rules

By applying one of the transformation rules on the graph of the start state, a new graph state is explored. These two graph states are connected by a *rule transition*, meaning the application of the rule on the start state yielding the new state. This structure resembles that of an LTS. In fact, by repeatedly applying all applicable transformation rules to each graph state until no new graph states can be explored, a transition system of graph states and rule transitions is found. This is called the *Graph Transition System* (GTiS) of the GTS. When the graph states and transition rules are labelled, for instance by numbering the states and giving the rules a label, the GTiS has the same structure as an LTS.

4.4 Example

The running example from Figure 4 is displayed as a GROOVE GTS model in Figure 5. Figure 5a is the start graph of the system. The rules can be described as follows:

1. 5b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 5c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 5d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The *LHS*, *NAC* and *RHS* of each rule are displayed as one graph. The colored nodes and edges in the rules indicate to which part they belong:

1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.
2. dotted line (red): This node or edge is part of the *NAC* only.
3. thick line (green): This node or edge is part of the *RHS* only.
4. dashed line (blue): This node or edge is part of the *LHS* only.

The *turn* flag on the **Player** node is a representation of a self-edge with label *turn*. The assignments on the **Die** node are representations of edges to integer nodes. The throws value assignment (:=) in the move rule represents an edge with the *throws* label to an integer node in the *LHS* and the same edge to another integer node in the *RHS*. In the next turn rule, the *turn* edge exists in the *LHS* as a self-edge of the left **Player** node and in the *RHS* as a self-edge of the right **Player** node. The *throws* edge from the left **Player** node to an integer node only exists in the *LHS*.

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 6 shows the GTiS of one *throws* rule application on the start graph. State s_1 is a simplified representation of the graph in Figure 5a. Figure 7 shows the graph represented by s_2.

5 Tooling

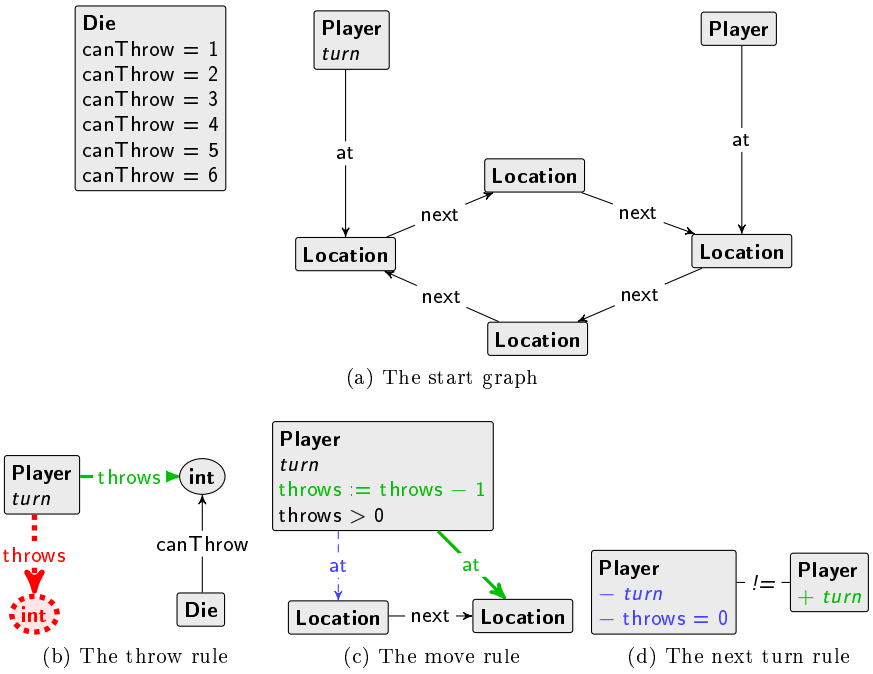
5.1 ATM

ATM is a web-based application, developed in the Ruby on Rails framework.

The architecture is shown graphically in Figure 8. It has a similar structure to the on-the-fly model-based testing tool architecture in Figure 2.

The tool functions roughly as follows:

Ik moet nog meer informatie over Axini en evt. oude klanten toevoegen



(a) The start graph

(b) The throw rule (c) The move rule (d) The next turn rule

Figure 5: The GTS of the board game example in Figure 4

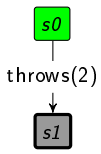


Figure 6: The GTiS after one rule application on the board game example in Figure 5

- An STS is given to a Model Interpreter component, which passes information on the current state and possible transitions to the Test Engine component, which devises the test cases.
- The stimuli given to the SUT are based on the labels of the switch relations of the STS. This component also includes the strategy component that decides on the specific data values to test.
- The Test Engine chooses a switch relation and the data values, thus a specific transition in the underlying LTS. This is given to the Test Execution component as an *abstract stimulus*. The term abstract is used here to indicate that the transition is specific to the model. It represents some computation steps taken in the SUT. For instance, the label 'connect' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT. This 'translation' is done by the Test Execution component. When the SUT responds, the Test Execution component translates this response back to an abstract response. The Test Engine updates the Model Interpreter on which transition(s) were chosen and gives the pass or fail verdict based on the test case.

5.2 GROOVE

The modelling tool GROOVE is in development at the University of Twente. It has been applied to several case studies, such as model transformations and security and leader election protocols [5].

The architecture of the GROOVE tool is shown graphically in Figure 9. A GTS is given as input to

Wat voor meer context/refs moeten er nog bij? Is er iets geschreven over dat ding met schoolkinderen?

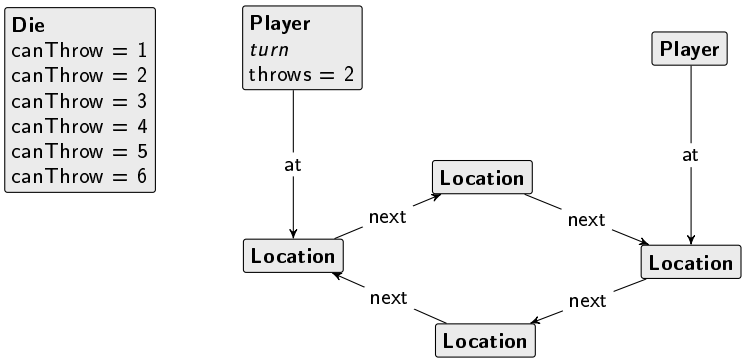


Figure 7: The graph of state $s2$ in Figure 6

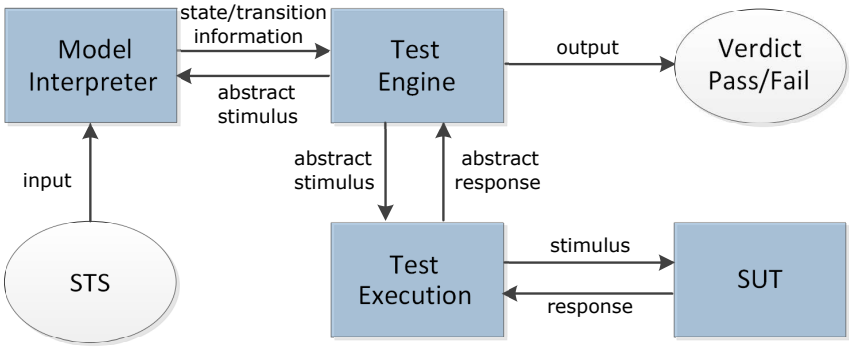


Figure 8: Architecture of ATM

a Rule Applier component, which determines the possible rule matches. An Exploration Strategy can be started or the user can explore the states manually using the GUI. These components request the rule matches and give the chosen match as feedback. The Exploration Strategy can do an exhaustive search, leading to a GTiS.

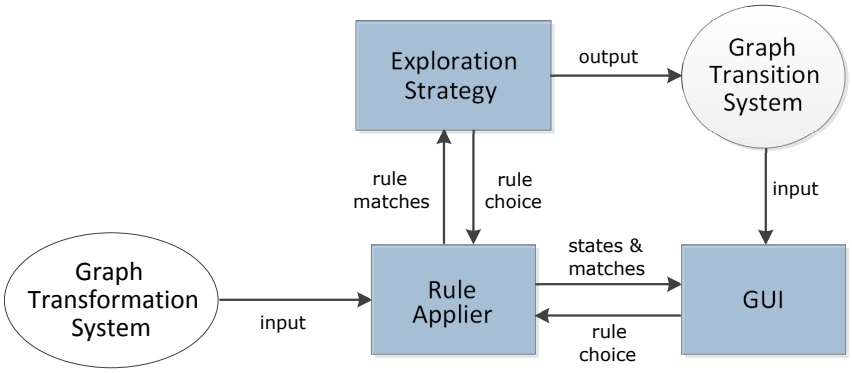


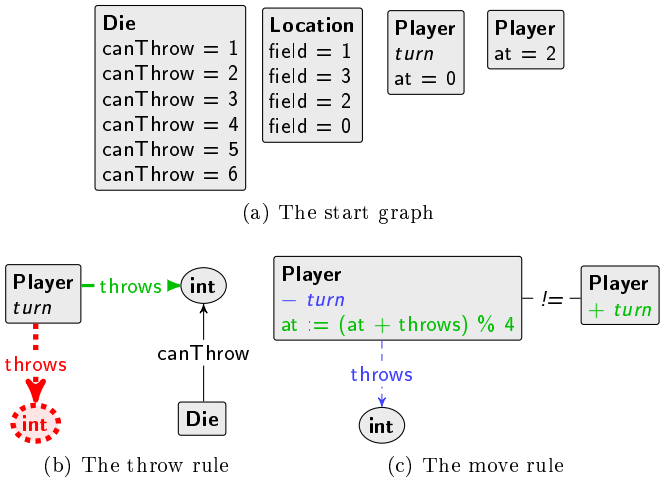
Figure 9: The GROOVE Tool

5.3 Comparison of the examples

The models of the boardgame example in Figures 5 and 4 are very different. However, the GTiS of the GTS can be found using GROOVE and the STS can be mapped to an LTS. In this section the differences between the GTS and the STS of the example are explored, using the GTiS of the

GTS and the LTS of the STS.

The GTS of the boardgame example has a number of consecutive transitions when a player moves. The *move* rule puts the **Player** on the next **Location** and lowers the remaining 'moves' by one. This is different from the STS, which updates the location variable in one transition. The GTS can also model the location as a variable and update this variable in one transition. This model is shown in Figure 10. The movement of the players as modelled in the GTS can also be modelled in an STS. Figure 11 shows this model.



(a) The start graph

(b) The throw rule

(c) The move rule

Figure 10: Another GTS model of the board game example

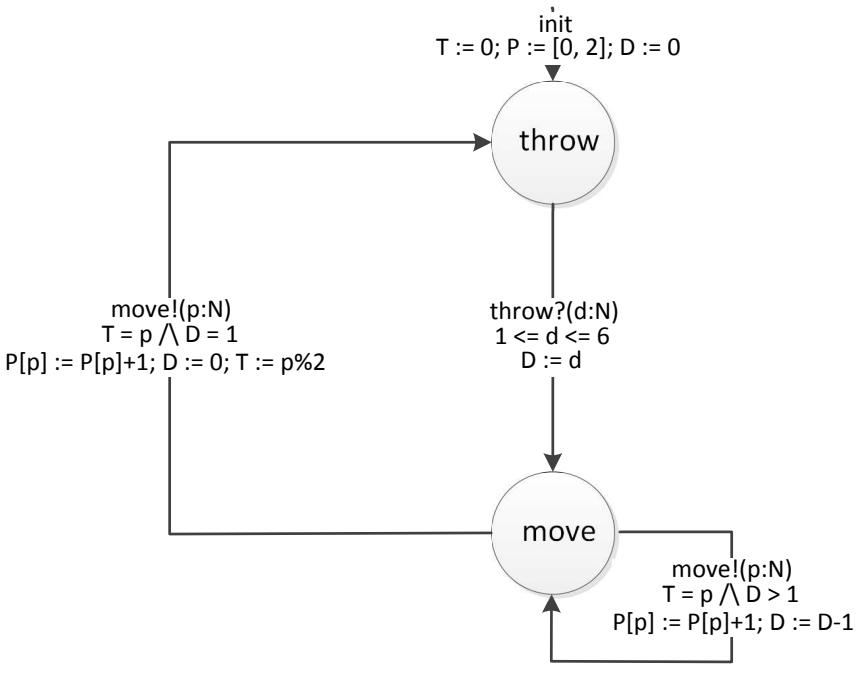


Figure 11: Another STS model of the board game example

The new GTS loses many advantages by structuring it in this way: the overview of the board is gone, the rules are less visual and extending the locations in different directions is much harder. The rules do look more compact, there is even one rule less. However, when generating the GTiSs of both models with GROOVE a significant difference appears: the GTS from Figure 5 generates

32 states with 52 transitions, which can be seen visually in Figure 12, whereas the model from Figure 10 generates 112 states with 192 transitions (Figure 13). The reason is that the board is circular: to the first model, the players being at locations 1 and 3 is the same as them being at locations 2 and 4. However, this is not the same to the second model. This distinction results in a *symmetry reduction* in the first model and thus a smaller statespace than of the second model. In both models, another symmetry reduction takes place when the players are at the same location.

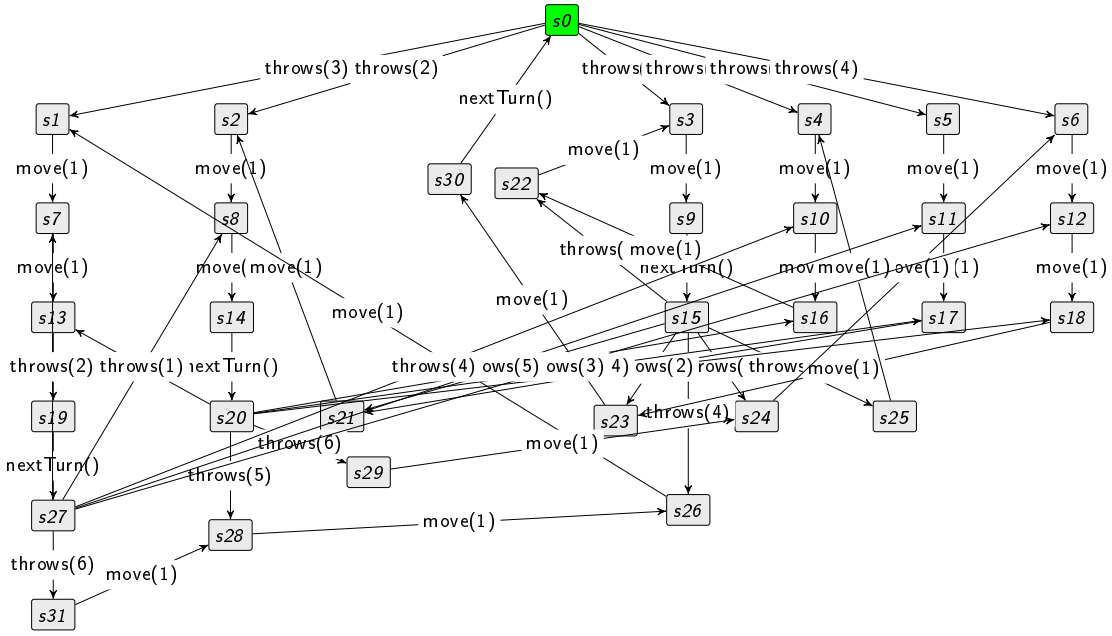


Figure 12: The GTiS of the model in Figure 5

The STS in Figure 11 is also different than the one in Figure 4: now two switch relations are needed to move a player. The underlying LTS of the first model has 224 states and 384 transitions. This is calculated by taking all possibilities of the data values except for the die roll. This leads to 32 states ($4 \times 4 \times 2$). These 32 'throw' states each have 6 throw transitions to a 'move' state, thus there are 192 'move' states. The 'move' states only have one transition back to a 'throw' state. There are $6 \times 32 + 192 \times 1 = 384$ transitions. The second STS does not have more states, because all possible combinations of data values have been taken into account. Each 'move' state still has one outgoing transition, to another 'move' state or back to the 'throw' state. Thus, for the STSs there is not much difference, except for example the observed behavior 'move! move!' instead of 'move!(2)'.

6 Research methods

6.1 Design

The specific method used to achieve the design goal is using GROOVE as a replacement of the symbolic model in ATM. Figure 14 shows this graphically. The Rule Applier component now communicates with the Model Interpreter component. The communication runs through interfaces on both tools, such that other tools like TorX are also able to communicate with the GROOVE component.

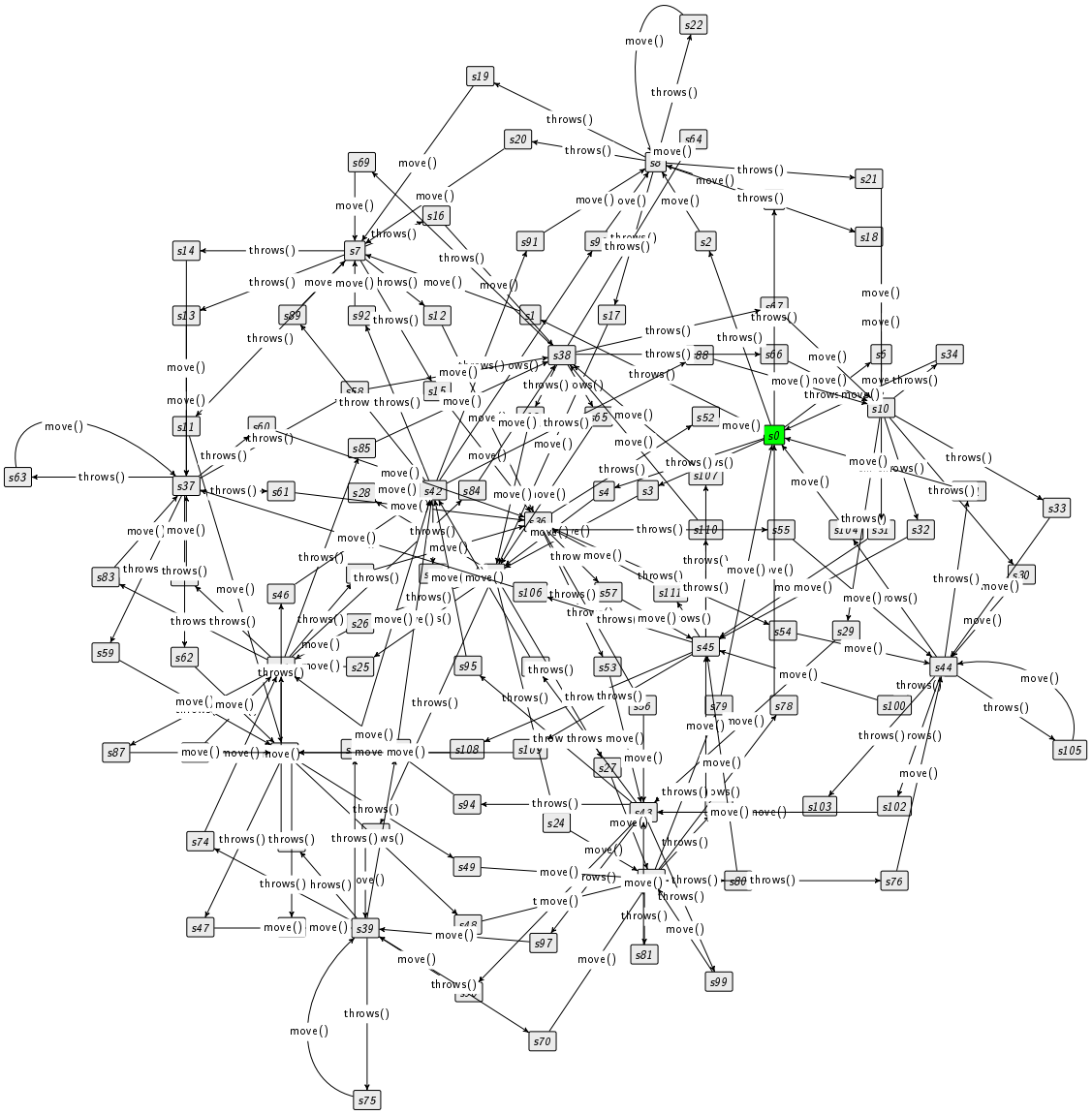


Figure 13: The GTiS of the model in Figure 10

6.1.1 Problems

Already some problems with the implementation have been identified:

- Modelling data types such as integers and strings in GROOVE is problematic; the range of possible values has to be given explicitly and cannot be infinite. For example, it would be impossible to extend the die of the running example such that it can throw any integer.
- Repeated application of a rule in a GTS is sometimes a sign of one transition spread out across multiple transitions. In the case of the running example, this occurs with the player moving step by step.
- ATM annotates the locations it has been and the switch relations it has taken in the STS. The GTS does not have these locations and switch relations (it does have states and transitions), so it is hard to annotate these on the GROOVE side. This annotation is used for coverage

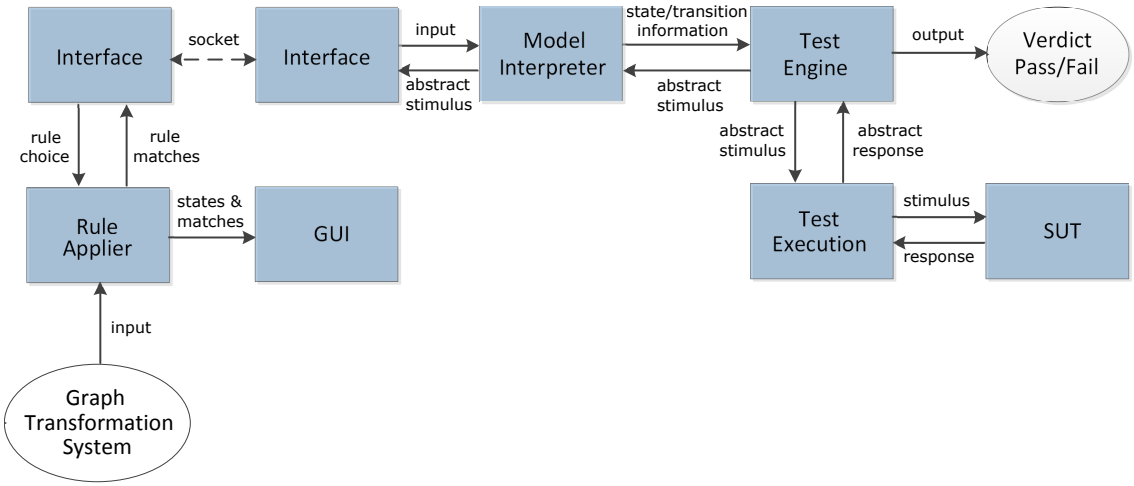


Figure 14: Replacing the symbolic model with GROOVE

statistics.

6.1.2 Coverage

State, transition, location and switch relation coverage are already implemented in the Axini tool. Location/switch relation coverage would also need to be implemented on the GROOVE tool. A representation of a location in GROOVE can be made by removing data values from a state graph. The applicable rules are a representation of the possible switch relations. Implementing this kind of 'graph/rule coverage' statistics is a possible field of research. An additional possible field of research is the implementation of data coverage statistics on ATM. This coverage statistic measures for each switch relation the representativeness of the data values used for all possible data values.

6.1.3 Design steps

The design steps for GRATiS will be done in increasing difficulty. In the next paragraph these steps are set out.

1. GROOVE will generate the entire LTS of the graph model and send it to ATM for the test generation. The test engine in ATM that accepts STSs is used, as this is the engine we want to modify. The LTS is also an STS, without variables, data values and contraints. The functionality of finding the underlying LTS of a GTS is already implemented in GROOVE. This will require the interface on both ends and the socket in between to be operational. The issue of limited data types should be solved at the end of this step.
2. The interface on the GROOVE side will translate the GTS directly to an STS and transmit that instead of the LTS. This will require GTS-to-STS transformation rules. The issue that can be solved here is the grouping of transitions.
3. The final step is to implement a GTS engine on ATM and communicate on-the-fly as the statespace of the GTS is explored on the GROOVE side. The final issue of location annotation for coverage statistics should be solved at the end of this step.

6.2 Validation

There are two steps in the validation:

1. The boardgame example is tested. A SUT is made for this game and ATM is used to test this game. Then, GRATiS is used to test the boardgame. The results should reveal no errors, fail verdicts or other differences in the output of both tools. An intentional error is then made in the SUT and the process is repeated. Still no errors or differences are expected, but both tools should find the error and give a fail verdict.
2. Next, the assessment of the strengths and weaknesses of GRATiS is done by applying both tools to several case studies and comparing the results. The case studies are set out first and then the criteria for the comparison are given.

6.2.1 Case studies

STSs or GTSs might prove more practical for one kind of system, while the other is more useful for another kind of system. Therefore, three case studies are planned. They are all real-life systems Axini has worked on:

- a self-scan register
- a navigation system
- a health-care system

The self-scan register is a machine that automates the purchase of products at a supermarket. A customer can put his products on a conveyer belt and the system automatically calculates the price of the products. Then the customer pays and gets a receipt. The navigation system is a GPS device with a route planner. It allows a user to enter a destination and the system plans the correct route accordingly from the location of the user. The health-care system is a medical device.

A GTS and an STS will be created for each system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the model and the test process will then be compared as part of the validation. The next two sections give the criteria for this comparison.

6.2.2 Objective criteria

As mentioned in section 1.2, the criteria for the comparison of ATM and GRATiS are split in two parts. The objective criteria that can be compared are:

1. The verdicts: same test cases should give same verdicts. When a verdict is different for the same test case in GRATiS and ATM, this might indicate an error.
2. The number of bugs found: one tool could generate smarter test-cases and find more bugs.
3. The coverage: generating smarter test-cases can also lead to a higher coverage.
4. The test cases generated per second: benchmarks will be done on how fast the tools generate test-cases.
5. The size of the statespace: benchmarks will be done on how much space the tools use.

6.2.3 Subjective criteria

This research will not include an extensive experiment with a statistical significant set of human actors. However, a group of experts in the field will be invited to a discussion at the end of the research period. This will include students and doctorates from the Formal Methods department of the University of Twente and employees from Axini. Preceding the discussion, the case studies and comparisions are presented. The discussion is meant for the participants to become acquainted with both modelling techniques and share ideas and thoughts. After the discussion, the participants are asked to work on one of the case studies. This work entails either:

1. Extending the model with a new feature
2. Finding a bug/error in the SUT/model

The following results are then noted and compared:

1. The time spent on the assignment
2. The correctness of the result
3. The feelings the participants had with the assignment (how difficult, how much fun, etc.)

With these results also the affiliation of the participants is taken into account: people from Axini and GROOVE are expected to do a better in their respective fields. Therefore, they will also be able to compare both modelling processes when working with the other tool than the one they are already familiar with.

The results will give insight in the understandibility, maintainability and extendibility of both modelling processes. The results are compiled and presented in the final thesis.

7 Summary

This report motivates the need of a research towards a model-based testing practice on Graph Transformation Systems. The research will investigate whether the assumption that GTSs will provide a more understandable, easier practice is true. The goal will be to create a tool that allows automatic test generation on GTSs and assess the strengths and weaknesses of this test practice.

The first observations in this report demonstrate that GTSs can provide a nice overview of a system. However, also the STSs are clear for this simple example. Increasing the complexity of the software system may change this. The transformation rules, given as separate graphs, provide a good overview of the behavior of the system. This feature should be beneficial in models for larger software systems.

The results also show an interesting automatic statespace reduction, namely the symmetry reduction. The second GTS of the example shows that also a purely arithmetic model can be built; this indicates the strength of the formalism.

The design phase is split into small steps that should break down the complexity of the entire implementation process. The validation with the use of the case studies emphasises the practicality of the tooling; the purpose of the test tools is to be used on real-world software systems. Finally, the experiments are a great indication of the usefulness of the tool towards software testers.

References

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-JÃűrg Kreowski, Sabine Kuske, Detlef Plump, Andy SchÃijrr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[3] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45510-8_9.

[4] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[5] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using groove. *International journal on software tools for technology transfer*, online pre-publication, March 2011.

[6] Hasan and Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311 – 325, 1992.

[7] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006. cited By (since 1996) 16.

[8] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7:297–315, 2005. 10.1007/s10009-004-0153-x.

[9] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[10] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[11] Joost-Pieter Katoen Manfred Broy, Bengt Jonsson and Alexander Pretschner. *Model-Based Testing of Reactive Systems*. Springer-Verlag Berlin Heidelberg, 2005.

[12] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29:55–64, July 2004.

[13] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[14] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[15] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[16] Jan Tretmans. A formal approach to conformance testing, 1992.

[17] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[18] Jan Tretmans. Model based testing with labelled transition systems. *Formal methods and testing: an outcome of the FORTEST network*, 2008.

A Planning

When	What	Deliverable
week 1-3 2 Jan - 22 Jan	Implementing basic interfaces for both tools. Creating SUT of boardgame example.	Working system for automatic test generation on GTSs by generating LTS.
week 4-6 23 Jan - 12 Feb	Setting up transformation rules for GTS-to-STS. Implementing transformation on GROOVE interface.	1) GTS-to-STS transformation rules. 2) System for automatic test generation on GTSs by generating STS.
week 7-10 13 Feb - 11 Mar	Implementing GTS engine on the Axini tool and graph/rule coverage on the GROOVE tool.	System for automatic on-the-fly test generation on GTSs with coverage statistics.
week 11-13 12 Mar - 1 Apr	Creating GTSs for each case study.	Graph-based models of several software systems.
week 14 2 Apr - 8 Apr	Running the implemented system on the created models.	Measurements and comparison of both tools on objective criteria.
week 15-16 9 Apr - 22 Apr	Optional: Researching possibilities of data coverage + implementation.	Support for data coverage statistics in both tools.
week 17-19 23 Apr - 13 May	Setting up experiments with participants.	Buggy models, case descriptions, appointments for experiments.
week 20 14 May - 20 May	Running experiments.	Measurements and comparison of both tools on subjective criteria.
week 21-23 21 May - 10 Jun	Finish writing thesis.	The final thesis.